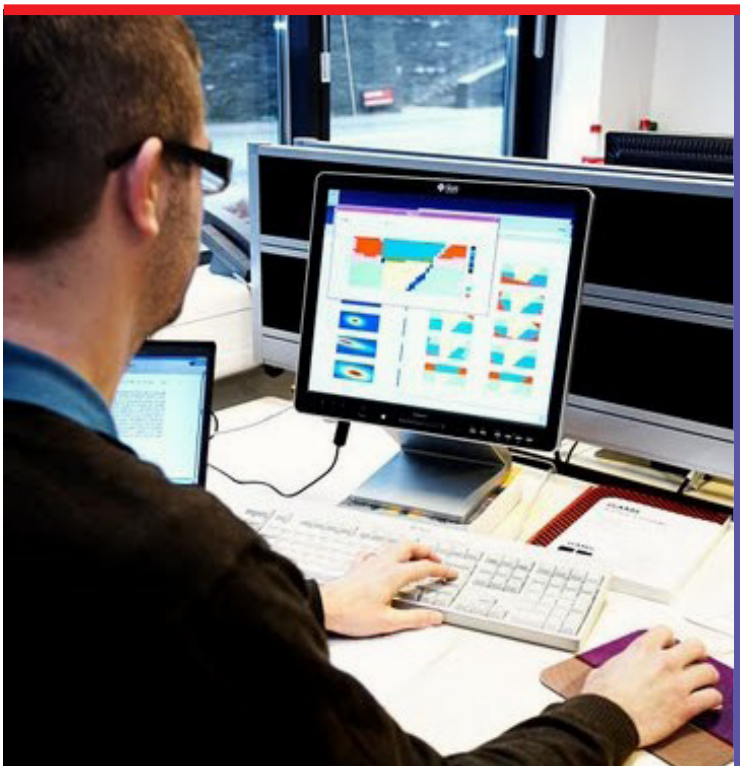
An Adaptive Large Neighborhood Search Algorithm for the Multi-mode RCPSP

Muller, Laurent Flindt

Publication date:
2011

Document Version
Publisher's PDF, also known as Version of record

An Adaptive Large Neighborhood Search Algorithm for the Multi-mode RCPSP



Report 3.2011

DTU Management Engineering

Laurent Flindt Muller
February 2011

DTU Management Engineering
Department of Management Engineering

An Adaptive Large Neighborhood Search Algorithm for the Multi-mode Resource-Constrained Project Scheduling Problem

Laurent Flindt Muller

Department of Management Engineering, Technical University of Denmark
Produktionstorvet, Building 426, DK-2800 Kgs. Lyngby, Denmark
lafm@man.dtu.dk

Abstract

We present an Adaptive Large Neighborhood Search (ALNS) algorithm for the Multi-mode Resource-Constrained Project Scheduling Problem (MRCPSP). We incorporate techniques for deriving additional precedence relations and propose a new method, so-called mode-diminution, for removing modes during execution. These techniques make use of bound arguments, and we propose and experiment with three new bounds for the MRCPSP, in addition to bounds found in the literature. We propose a simple technique, so-called opportunistic mode-flipping, which can be applied whenever a schedule is generated, and which significantly improves the results of the algorithm. Computational experiments are performed on a set of standard benchmark instances from the PSPLIB, and a comparison is made with other algorithms found in the literature. The experiments show that the algorithm is competitive, but can not beat the best algorithms. Even so, some of the elements of the algorithm perform well, that is the bound arguments, the mode-removal procedure, and in particular opportunistic mode-flipping, and these elements may perhaps be used to improve the results of other algorithms for this problem.

1 Introduction

Many processes within production scheduling and project management consists of scheduling a number of activities, each activity having a certain duration and requiring a certain amount of limited resources. Resources could for instance be machines or labor. Precedence relations may exist between activities, such that one activity can not start before others are completed. Typically one wishes schedule to the activities such that the total time taken to complete them is minimized. Such problems can be modeled as a Resource-Constrained Project Scheduling Problem (RCPSP), which is a generalization of the well-known Job-Shop-Scheduling problem. There exists a number of variants of the RCPSP, see for instance Blażewicz et al. (1983), Brucker et al. (1999), or Hartmann and Briskorn (2009). We consider the the Multi-mode Resource-Constrained Project Scheduling Problem (MRCPSP), which is a popular variant. In the MRCPSP each activity can be performed in a number of different so-called modes, each representing alternative ways of executing the activity.

There exists a large body of work for the single-resource RCPSP (SRCPSP) and to a lesser extend for the MRCPSP. We refer to the surveys by Hartmann and Drexl (1998), Herroelen et al. (1998), Kolisch and Hartmann (2000, 2006), Kolisch and Padman (2001), and Patterson (1984). Among the heuristic solution methods for the MRCPSP are the immune-system based algorithm of Van Peteghem and Vanhoucke (2009), the (hybrid) genetic algorithms of Alcaraz et al. (2003), Hartmann (2001), Lova et al. (2009), Mori and Tseng (1997), Okada et al. (2010), Ozdamar (1999), Tseng (2008) and Tseng and Chen (2009), the tabu-search based algorithm of Tchao and Martins (2008), the simulated annealing-based algorithms of Bouleimen and Lecocq (2003) and Józefowska et al. (2001), the ant-colony optimization based algorithm of Chiang et al. (2008), the particle swarm optimization algorithm of Jarboui et al. (2008) and Zhang et al. (2006),

the differential evolution based algorithm of Damak et al. (2009), the hybrid scatter search based algorithm of Ranjbar et al. (2009), and the local-search based algorithms of Boctor (1996), Drexl and Gruenewald (1993) and Kolisch and Drexl (1997). Currently the best performing algorithms are the immune-system based algorithm of Van Peteghem and Vanhoucke (2009) and the genetic algorithm of Lova et al. (2009).

In this paper we experiment with an Adaptive Large Neighborhood Search (ALNS) algorithm, which is a heuristic procedure. The motivation for using an ALNS based algorithm is twofold: (1) ALNS is a relatively new meta-heuristic approach, which has been applied with good results within the context of vehicle routing problems, and it would be of interest to see how the approach performs in a different context. (2) The ALNS framework is very flexible, making it possible to incorporate different neighborhood structures and letting the adaptive layer of the ALNS algorithm select among the best performing during execution. The hope is that using this approach to incorpore neighborhoods from state-of-the-art algorithms for the RCPSP into a single algorithm, will be beneficial. A similar approach was examined by Muller (2009) for the single-mode RCPSP with moderate success, and the algorithm described here is an extension of the algorithm of Muller (2009).

We incorporate techniques for deriving additional precedence relations and propose and experiment with a new method for removing modes during execution (so-called mode-diminution). These techniques make use of bound arguments, and we propose and experiment with three new bounds (named LBX1, LBX2, and LB2X) for the MRCPSP, in addition to bounds found in the literature. A preprocessing step, which strengthens these bounds, is also proposed (so-called resource strengthening). We finally propose and experiment with a simple technique which can be applied every time a new schedule is generated, and which significantly improves the results (so-called opportunistic mode-flipping). The technique can be seen as an extension of the single-pass improvement method used by Hartmann (2001) for his genetic algorithm.

Computational experiments are performed on a set of standard benchmark instances from the PSPLIB, and a comparison is made with other algorithms from the literature. These experiments show that the proposed algorithm is competitive, but can not beat the best found in the literature. Even so, mode-diminution and in particular opportunistic mode-flipping perform well, and the new bounds improve upon existing ones. Some of these elements may perhaps successfully be used to improve upon solutions for other heuristics for this problem. To the best of our knowledge, this is the first application of an ALNS algorithm to the MRCPSP.

The remaining of the document is organized as follows: in Section 2 a formal description of the RCPSP is given, in Section 3 the ALNS framework is described, in Section 4 a number of lower bounds used in the algorithm are described, in Section 5 the algorithm is described, in Section 6 the neighborhoods are described, in Section 7 the computational experiments are presented, and we conclude in Section 8.

2 Problem description

The MRCPSP can be described as follows (see for instance Brucker et al. (1999)): A project consists of a set $\mathcal{A} = \{1, \ldots, n\}$ of *activities*, which must be scheduled. Traditionally activity 1 and n are so-called dummy activities, which represent the start and the end of the project. Let $\mathcal{A}^* = \mathcal{A} \backslash \{1, n\}$. Each activity j can be performed in a number of different *modes* $\mathcal{M}_j = \{1, \ldots, |\mathcal{M}_j|\}$, each representing a different way of performing the activity. There are two sets of resources, (1) *renewable resources* $\mathcal{R} = \{1, \ldots, |\mathcal{R}|\}$, and (2) *nonrenewable resources* $\tilde{\mathcal{R}} = \{1, \ldots, |\tilde{\mathcal{R}}|\}$. A renewable resource $k \in \mathcal{R}$, has capacity R_k in each time period, while a nonrenewable resource $k \in \tilde{\mathcal{R}}$ has capacity $\tilde{R}_k$ across all time periods. When an activity j is scheduled in mode $m \in \mathcal{M}_j$, it has a *processing time* of p_{jm} (non-preemtible) and requires $r_{jkm} \geq 0$ units of renewable resource $k \in \mathcal{R}$ in each time period, and $\tilde{r}_{jkm} \geq 0$ of nonrenewable resource $k \in \tilde{\mathcal{R}}$ across all time periods. There exists *precedence relations* between the activities, such that one activity $j \in \mathcal{A}$ can not be started before all its predecessors, $\mathcal{P}_j$, are completed. Symmetrically $\mathcal{S}_j$ denotes the set of successors and $\mathcal{N}_j = \mathcal{A} \backslash (\mathcal{P}_j \cup \mathcal{S}_j)$ denotes the set of activities which can run in parallel with j. Let

$\mathcal{E} = \{(i,j) \in \mathcal{A} \times \mathcal{A} : i \in \mathcal{P}_j\}$ be the set of all precedence relations. Let $\mathcal{T}$ be a set of time-steps during which the activities must be performed. The MRCPSP may be formulated as follows:

$$
\begin{aligned}
&\min \ \sigma_n \\
&s.t. \ \sigma_j \geq \sigma_i + p_{i,m(i)} && \forall j \in \mathcal{A}, \forall i \in \mathcal{P}_j && (1)\\
&\sum_{j \in A(t)} r_{j,k,m(j)} \leq R_k && \forall k \in \mathcal{R}, \forall t \in \mathcal{T} && (2)\\
&\sum_{j \in \mathcal{A}} \tilde{r}_{j,k,m(j)} \leq \tilde{R}_k && \forall k \in \tilde{\mathcal{R}} && (3)\\
&\sigma_j \geq 0 && \forall j \in \mathcal{A}, && (4)
\end{aligned}
$$

where σ_j is the starting time of activity j, $m(j)$ is mode chosen for activity j, and $A(t) = \{j \in \mathcal{A} | \sigma_j \leq t \leq \sigma_j + \pi_j\}$, i.e., the activities in progress at time t. Constraints (1) are denoted the *precedence constraints*, constraints (2) are denoted the *renewable resource constraints* (or *resource constraints* for short), and constraints (3) are denoted the *non-renewable resource constraints*. As mentioned previously the RCPSP (and thus also the MRCPSP) is a generalization of the Job Shop Scheduling Problem and is therefore $\mathcal{NP}$-hard (see e.g. Blażewicz et al. (1983)).

3 Adaptive large neighborhood search

The ALNS framework was first proposed by Pisinger and Røpke (2007) for solving different variants of vehicle routing problems, where good results have been reported by Pisinger and Røpke (2007) and Røpke and Pisinger (2006a,b). It is a framework which can be applied to a large class of optimization problems and can be described as follows: ALNS is a local search framework in which a number of simple neighborhoods compete to modify the current solution. In each iteration a *destroy neighborhood* is chosen to destroy the current solution, and a *repair neighborhood* is chosen to repair the solution. The new solution is accepted if it satisfies some criteria defined by the local search method applied at the master level. The neighborhoods used are typically large neighborhoods, which can reach a large part of the solution space.

An adaptive layer stochastically controls which neighborhoods to choose based on their past performance (score). The more a neighborhood has contributed to the quality of the current solution, the larger score it obtains, and hence it has a larger probability of being chosen. The adaptive layer uses roulette wheel selection for choosing one of the destroy and one of the repair neighborhoods. If the past score of a neighborhood i is π_i and we have ω neighborhoods, then we choose neighborhood j with probability $\pi_j / \sum_{i=1}^{\omega} \pi_i$. ALNS can be based on any local search method, e.g., simulated annealing, tabu search or guided local search. An outline of the ALNS framework is given by Algorithm 1, for a more detailed description we refer to Pisinger and Røpke (2007).

4 Lower bounds

Good lower bounds are an important component in branch-and-bound algorithms in order to be able to prune the search tree effectively. They may also prove useful in a heuristic setting, where they may be used to speed up the heuristic by terminating early when an optimal solution has been found, or to restrict the heuristic to parts of the solution space, where improving solutions may be found. In the following we first give an description of the concept of a Finish-to-Start-Distance (FSD)-matrix, and then the bounds examined in connection with the algorithm.

Finish-to-Start-Distance (FSD)-matrix We employ a slight modification of the FSD-matrix of Zhu et al. (2006), which is in itself a modification of the traditional Start-to-Start distance

Algorithm 1 Outline of the ALNS framework

```
1: Construct feasible solution x.
2: x* ← x
3: while stop criteria not met do
4:    Choose a destroy neighborhood N⁻ and repair neighborhood N⁺ using roulette wheel selection based on previously obtained scores {π_j}.
5:    Generate a new solution x' from x using the heuristics corresponding to the chosen destroy and repair neighborhoods, N⁻ and N⁺.
6:    if x' can be accepted then
7:       x ← x'
8:    end if
9:    Update scores π_j of N⁻ and N⁺.
10:   if f(x) < f(x*) then
11:      x* ← x
12:   end if
13: end while
14: return x*
```

matrix found in the literature, see for instance Bartusch et al. (1988), Brucker and Knust (2000), or Demassey et al. (2005). An FSD-matrix is an integer matrix, $B = (b_{ij})_{\mathcal{A}\times\mathcal{A}}$, which satisfies:

$$\sigma_j - \tau_i \geq b_{ij} + 1, \quad \forall (i,j) \in \mathcal{A} \times \mathcal{A}$$

for all feasible schedules, where τ_i is the completion time of activity i and σ_j is the starting time of activity j. That is b_{ij} is a lower bound on the amount of time which must pass between the completion time of i and the starting time of j. Note that b_{ij} may be negative, which means that activity j must start before the completion of activity i. Define $\underline{p}_i := \min\{p_{im} | \, m \in \mathcal{M}_i\}$ and $\overline{p}_i := \max\{p_{im} | \, m \in \mathcal{M}_i\}$. Since the relation (4) has the following transitive property:

$$\sigma_j - \tau_i \geq b_{ij} + 1 \wedge \sigma_k - \tau_j \geq b_{jk} + 1 \Rightarrow \sigma_k - \tau_i \geq b_{ij} + \underline{p}_j + b_{jk} + 1,$$

the entries of an FSD-matrix may be updated by calculating the transitive closure of B. This can be done using a variant of the Floyd-Warshall algorithm in $O(|\mathcal{A}|^3)$ time, see Zhu et al. (2006). Note that a pair of activities (i,j) must run in parallel if $b_{ij} \geq -\underline{p}_i - \underline{p}_j + 1$ and $b_{ji} \geq -\underline{p}_i - \underline{p}_j + 1$. Given an upper bound, T, on the makespan the FSD-matrix B may be initialized as follows:

$$b_{ij} = \begin{cases} -\overline{p}_i & \text{if } i = j \\ 0 & \text{if } (i,j) \in \mathcal{E} \\ -T & \text{otherwise,} \end{cases}$$

The difference between the FSD-matrix considered here, and the one of Zhu et al. (2006) is that here the FSD-matrix is defined for all pairs of activities, whereas it in Zhu et al. (2006) is defined only for pairs of activities being related by precedence.

As will be explained in detail in Section 5, an FSD-matrix may be used to discover new precedence relations, to eliminate modes from consideration, and to prove optimality. The effectiveness of the methods rely on good bound arguments for the b_{ij} values. The values b_{ij} may be calculated by constructing the subproblem defined by $\mathcal{S}_i \cap \mathcal{P}_j$ and then applying any lower bounding technique for the RCPSP on this smaller instance. A number of such lower bounds exists in the literature. We do not give a description of these here but instead refer the reader to the excellent work by Klein and Scholl (1999) where 11 such bounds are described and compared. Using the name convention of Klein and Scholl (1999), the bounds considered here are: LB1, LB2, LB6, LB8, LB10, and LB11. LB1 is the well-known critical path lower bound, LB2 is the also well-known capacity bound (explained in more detail below), LB6 and LB8 are extensions of the so-called weighed node packing bound of by Mingozzi et al. (1998) (see Klein and Scholl (1999)), and LB10 and

LB11 are bounds based on evaluating a lower bound for all possible ways of resolving a resource conflict given respectively pairs, and triples of activities (again see Klein and Scholl (1999)).

The bounds LB6, LB10, and LB11 again use lower bound arguments on subproblems of the problem considered. Note that this problem may in itself be a subproblem corresponding to a b_{ij} entry. We will refer to the former as *inner* subproblems and the latter as subproblems. When calculating bounds on inner subproblems, one could recursively apply bound arguments, but this can potentially by very time-consuming, and we thus, like Klein and Scholl (1999), only apply the critical path lower bound (LB1) and capacity bound (LB2). For the computational experiments we will investigating replacing LB2 with an extended variant (LB2X described below), which provides better bounds, but at the cost of additional running time.

In the following we describe three new bounds (LBX1, LBX2, and LB2X) for the MRCPSP. These bounds are improvements of the critical path lower bound (LB1), and capacity lower bound (LB2), and we begin with a short description of the latter.

LB1 (critical path) This bound is computed by finding the longest path (also called the critical path) in the precedence graph. For the multi-mode case this critical path is calculated based on the minimum processing time of each activity.

LB2 (capacity bound) For a activity $i \in \mathcal{A}$, define $a_{ikm} := p_{im}r_{ikm}/R_k$ and let $\underline{a}_{ik} = \min_{m\in\mathcal{M}_i}\{a_{ikm}\}$. This bound is calculated as

$$LB2 = \max_{k\in\mathcal{R}}\left\{\left\lceil\sum_{i\in\mathcal{A}}\underline{a}_{ik}\right\rceil\right\},$$

i.e., it is a lower bound based on the amount of renewable resource available versus the amount of renewable resource required by a set of activities.

LBX1 (mode-fixed critical path 1) Let I be some multi-mode instance. Let $LB1(I)$ and $LB2(I)$ denote the lower bounds as computed by the bounds LB1 and LB2 above. For a subset $S \subseteq \mathcal{A}$ let $\mathcal{M}_S$ denote the set of feasible mode assignments for the activities of S. For a feasible mode assignment $m \in \mathcal{M}_S$, let $I(m)$ denote the corresponding restriction of I to the selected modes. It is clear that the lower bound calculated as

$$LB = \max_{S\in\mathcal{S}}\min_{m\in M_S}\max\{LB1(I(m)), LB2(I(m))\},$$

where $\mathcal{S}$ is some powerset of $\mathcal{A}$, is a strengthening of LB1 and LB2 in the multi-mode case. As the number of feasible mode assignments grows exponentially in the size of S, S must be kept small in order for the bound to be computationally tractable. LBX1 is calculated as above, where $\mathcal{S}$ is the set of all pairs of activities from $\mathcal{A}$.

LBX2 (mode-fixed critical path 2) LBX1 may be improved by considering larger subsets of activities. LBX2 is calculated as LBX1, except $\mathcal{S}$ is the set of all pairs, and triples of activities of $\mathcal{A}$.

LB2X (extended capacity bound) The capacity bound LB2 above is calculated separately for each resource. In the multi-mode case the bound may be strengthened by taking into account all the resources simultaneously. This amounts to solving the following Mixed Integer Programming

(MIP) problem:

$$\min\ b \tag{5}$$

$$s.t. \sum_{i\in\mathcal{A}}\sum_{m\in\mathcal{M}_i} a_{ikm}x_{im} \le b \qquad \forall k \in \mathcal{R} \tag{6}$$

$$\sum_{m\in\mathcal{M}_i} x_{im} \ge 1 \qquad \forall i \in \mathcal{A} \tag{7}$$

$$b \ge 0 \tag{8}$$

$$x_{im} \in \{0,1\} \qquad \forall i \in \mathcal{A}, m \in \mathcal{M}_i \tag{9}$$

where $x_{im} = 1$ if and only if activity i uses mode m.

We now show that this problem is $\mathcal{NP}$-hard by reduction from the $\mathcal{NP}$-hard partition problem (see Karp (1972)), which in its optimization version asks: Given a set C of integer numbers, find a partition of C into two subsets C_1 and C_2 such that $\max(sum(C1), sum(C2))$ is minimized. The reduction is as follows: Let $|\mathcal{R}| = 2$ and let each resource have unit capacity. For each $c \in C$, create an activity i with two modes with resource usages $(c, 0)$ and $(0, c)$.

We thus consider a Lagrangian relaxation, where the constraints (6) are dualized. This gives rise to the following Langragian Dual (LD) problem:

$$\max_{\substack{\lambda_k\ge 0\\ ,\sum_{k\in\mathcal{R}}\lambda_k\le 1}} \quad \min \sum_{k\in\mathcal{R}} \lambda_k \sum_{i\in\mathcal{A}}\sum_{m\in\mathcal{M}_i} a_{ikm}x_{im} \tag{10}$$

$$s.t. \sum_{m\in\mathcal{M}_i} x_{im} \ge 1 \qquad \forall i \in \mathcal{A} \tag{11}$$

$$x_{im} \in \{0,1\} \qquad \forall i \in \mathcal{A}, m \in \mathcal{M}_i, \tag{12}$$

For any choice of λ the problem may be solved in linear time. Note that if one chooses $\lambda_k = 1$ and $\lambda_{k'} = 0\,\forall k' \neq k$, then the solution to the inner minimization problem is the same as LB2 calculated on the resource k. The LD problem can be solved by using a subgradiant algorithm.

LB2X is defined as the best solution found after a specified number of iterations of the subgradiant algorithm. The number of iterations has been experimentally fixed to 40.

As we will see later applying subgradiant algorithm may prove too slow in certain cases, when the bound is used for the inner subproblems. We thus define an additional bound, LB2X', which is faster to compute, but also weaker: the bound is defined as the best solution to the LD problem for a fixed number of of values of λ ($|\mathcal{R}|+1$ in total). These values of λ are $\lambda_k = 1$ and $\lambda_{k'} = 0\,\forall k' \neq k$, where $k = 1 \ldots |\mathcal{R}|$ (giving $|\mathcal{R}|$ values of λ), and $\lambda_k = 1/|\mathcal{R}|\,\forall k \in \mathcal{R}$ (giving one additional value of λ).

5 Algorithm

The algorithm proposed is an extension to the MRCPSP of the algorithm presented in Muller (2009) for the single-mode RCPSP. For the sake of completion we here repeat the elements from that algorithm, which are relevant for the extension, and describe some new elements specific to the extension.

Representation A number of different solution representations of the RCPSP has been proposed in the literature (see for instance Kolisch and Hartmann (1999)). The proposed algorithm employs list representation, where a schedule is represented as an ordered list of activities. The ordering is as follows: Let $(i, j) \in \mathcal{E}$ and let γ_i and γ_j be the positions within the list of activity i and j respectively, then $\gamma_i < \gamma_j$.

When employing list representation, one additionally needs a scheme for converting the list into a schedule. The two most commonly used are the serial and parallel Schedule Generation Schemes (SGSs) (see for instance Kolisch and Hartmann (1999)). The serial SGS can be described

as follows: in the order defined by the list, schedule each activity in turn, at the earliest precedence and resource feasible point in time. The parallel SGS can be described as follows: Time is incremented starting at zero. At each time the unscheduled precedence and resource feasible activities are scheduled in the order defined by the list. When no more activities can be scheduled at the current time, the time is incremented. For the ALNS algorithm examined in Muller (2009), in context of the SRCPSP the serial SGS turned out to produce better solutions than the parallel SGS and we therefore used the serial SGS here also.

Local search method As noted in Section 3 one needs to select a local search method on top of which to build the ALNS algorithm. The method used here is as follows: In each iteration a destroy and repair neighborhood is selected based on the current scores. Given the parameter Q, which governs how many activities may be moved within the current list, a new schedule (and corresponding activity list) is created based on the neighborhoods selected. Only solutions which are as good as, or better than, the current solution are accepted. A tabu list is maintained to ensure that the same activity list is not visited twice. The value Q is progressively reduced from its initial value toward a final low value. This has the effect that in the beginning the algorithm will look at large neighborhoods, but as the search progresses to good solutions the size gets progressively reduced. The idea is that when a good solution has been found, the overall structure of the solution is sound and should be kept but small changes may find a new better solution.

Scoring scheme As part of an ALNS algorithm, a scoring scheme for the destroy and repair neighborhoods needs to be decided upon. As in Pisinger and Røpke (2007) the scores are updated at certain intervals rather than in each iteration. Thus scoring information is collected during a certain number of iterations before the scores of each neighborhood is updated and the collection restarts. Following the naming of Pisinger and Røpke (2007), we call the number of iterations which must pass between each score update the *score interval*. Let π_j be the current score, i.e., the score on the basis of which neighborhoods are chosen, and let $\bar{\pi}_j$ the score accumulated during the score interval (it will be explained later how $\bar{\pi}_j$ is accumulated), then the score π_j is updated as follows at the end of a score interval: $\pi_j = \max\{\rho \cdot \frac{\bar{\pi}_j}{a_j} + (1-\rho) \cdot \pi_j, \pi_{min}\}$, where a_j is the number of times the neighborhood has been chosen during the last score interval (if $a_j = 0$, the score remains identical to the score in the previous interval), ρ is the *score reaction* and π_{min} is the *minimum score*.

Let $|S_x|$ be the makespan of the current solution, $|S_{x^*}|$ the makespan of the current best solution, and $|S_{x'}|$ the makespan of the current candidate created by applying the destroy and repair neighborhood N^+ and N^-. Let $\bar{\pi}_j$ be the collected score so far for N^+ (the procedure is equivalent for N^-), $\bar{\pi}_j$ is updated as follows:

$$\bar{\pi}_j = \bar{\pi}_j + \begin{cases} 10 & \text{if } |S_{x'}| < |S_{x^*}| \\ 5 & \text{if } |S_{x'}| < |S_x| \\ 2^{|S_x|-|S_{x'}|-1} & \text{otherwise} \end{cases}$$

This scoring scheme differs slightly from the one employed in Pisinger and Røpke (2007), where one of three fixed scores are attributed depending on whether the current solution was improved globally, locally or a worse solution was accepted. The reason for this difference is that the proposed algorithm only accepts equal or better solutions (while the local search method employed in Pisinger and Røpke (2007) may accept worse solutions), which can results in many iterations where there is no improvement at all. If a fixed scoring scheme were used all neighborhoods would be scored equally bad in such a situation, while for the employed scheme it is possible to differentiate the neighborhoods who produce solutions which are (almost) as good as the current and the ones that produce solutions which are far worse. Note that even though only better solutions are accepted, the middle case above may occur because of the mode-change-algorithm, precedence augmentation, and mode diminution.

Modes Throughout the course of the algorithm a feasible mode-selection is maintained. Note that finding a feasible mode selection is in itself an $\mathcal{NP}$-hard problem. In the initial stage of the algorithm a feasible mode-selection is found by applying the Minimum Normalized Resources (MNR) method of Lova et al. (2009). For the few cases, where this method can not find an feasible mode-selection, we apply exhaustive search.

The destroy and repair neighborhoods work only on the current mode-selection and are oblivious to other modes. At regular intervals a mode-change-algorithm is run, which attempts to find a new mode selection, which then become the current until the next call to the mode-change-algorithm. Let UB be the current upper bound, then the new mode-selection should be such that the lower bound does not exceed $UB - 1$, otherwise no improving solution can be found. If the optimal solution has been found, it may happen that no such mode-selection exists. In order not to spend too much time searching for mode-selections with $LB \leq UB - 1$, the frequency of calls to the mode-change-algorithm is halved when no mode-selection could be found. Initially this frequency is set to 2, i.e., every other iteration of the ALNS algorithm. The mode-change-algorithm can be seen in Algorithm 2.

Algorithm 2 Pseudocode for the mode-change-algorithm

```
for all distinct triples (i, j, k) ∈ A × A × A do
  for all mode combinations (m_i, m_j, m_k) ∈ M_i × M_j × M_k do
    if change to (m_i, m_j, m_k) is feasible then
      Make change, and calculate lower bound: LB.
      if LB ≤ UB − 1 then
        Double call-frequency (if previously halved).
        return new mode-selection
      end if
      Proceed with a new triple.
    end if
  end for
end for
Halve call-frequency.
return new mode-selection (does not satisfy LB ≤ UB − 1)
```

Initial mode and resource reductions As described by Sprecher et al. (1997) the number of modes and resources may be reduced by application of the following preprocessing procedure: Define $\underline{\tilde{r}}_{ik} := \min\{\tilde{r}_{ikm} | \, m \in \mathcal{M}_i\}$, and $\overline{\tilde{r}}_{ik} := \max\{\tilde{r}_{ikm} | \, m \in \mathcal{M}_i\}$. A mode $m \in \mathcal{M}_i$ is called *non-executable* if either (1) $r_{ikm} > R_k$, for some $k \in \mathcal{R}$, or (2) $\sum_{j \in \mathcal{A} \setminus \{i\}} \underline{\tilde{r}}_{jk} + r_{ikm} > \tilde{R}_k$, for some $k \in \tilde{\mathcal{R}}$. A mode is called *inefficient* if there exits another mode m' of $\mathcal{M}_i$, such that $r_{ikm} \geq r_{ikm'} \, \forall k \in \mathcal{R}$, and $\tilde{r}_{ikm} \geq \tilde{r}_{ikm'} \, \forall k \in \tilde{\mathcal{R}}$. A non-renewable resource k is called redundant if $\sum_{i \in \mathcal{A}} \overline{\tilde{r}}_{ik} \leq \tilde{R}_k$. Non-executable and inefficient modes, and redundant non-renewable resources can be removed initially by the use of the algorithm described in Algorithm 3. This procedure is also used by Alcaraz et al. (2003), Hartmann (2001), Józefowska et al. (2001), Okada et al. (2010), Ranjbar et al. (2009), and Tseng and Chen (2009) for their algorithms.

Algorithm 3 Pseudocode for preprocessing of modes and non-renewable resources

```
Remove non-executable modes.
repeat
  Remove redundant non-renewable resources.
  Remove inefficient modes.
until No mode removed
```

Initial resource strengthening The lower bounds LB2, LB2X, and LB2X' are dependant upon the resource usages. If the resource usage of an activity in a certain mode can be increased without changing the optimal solution, the resulting bounds will be stronger. The following rule is thus applied after the initial mode and resource reductions: Let $(i, j) \in \mathcal{A} \times \mathcal{A}$ be a pair of activities. We say that two modes $m_i \in \mathcal{M}_i$ and $m_j \in \mathcal{M}_j$ are *incompatible* if they can not be run in parallel, either because of precedence relations between the activities, or because of resource constraints. If a mode $m \in \mathcal{M}_i$ of an activity $i \in \mathcal{A}$, is found, which is incompatible with all other modes of all other activities, then the resource usage is updated as $r_{ikm} := R_k \, \forall k \in \mathcal{R}$.

Precedence augmentation When a new upper bound, UB, is found, one is only interested in finding better solutions. Thus any sequencing of activities which has a lower bound larger than $UB - 1$ can be forbidden. Let the *head*, h_j, of an activity $j \in \mathcal{A}$ be the time that must pass before activity j can be started and let the *tail*, t_j, be the time that must pass from the point where activity j has completed until the project can be completed. The head and tail of an activity j can respectively be read from the entries b_{0j} and b_{jn} of the FSD-matrix. Define $\underline{r}_{ik} := \min\{r_{ikm} | \, m \in \mathcal{M}_i\}$. We now describe how precedence relations may be deduced on the basis of heads and tails.

The following two rules (13) and (14), are a simple generalization to the multi-mode case of a subset of the rules employed by Fleszar and Hindi (2004) in their variable neighborhood search algorithm. Let $i, j \in \mathcal{A}$ be a pair of activities for which no precedence relations exists. Precedence relations may be deduced as follows:

$$h_j + t_i \geq UB \Rightarrow i \rightarrow j \tag{13}$$

$$\exists k \in \mathcal{R} : \underline{r}_{ik} + \underline{r}_{jk} > R_k \wedge h_j + \underline{p}_j + \underline{p}_i + t_i \geq UB \Rightarrow i \rightarrow j. \tag{14}$$

We additionally use the following deduction rule: Let $i, j, k \in \mathcal{A}$ be a triple of activities for which no precedence relation exists and assume that none of the three can be run in parallel. We examine all 6 sequencing possibilities of the three activities: (1) $i \rightarrow j \rightarrow k$, (2) $i \rightarrow k \rightarrow j$, (3) $j \rightarrow i \rightarrow k$, (4) $j \rightarrow k \rightarrow i$, (5) $k \rightarrow i \rightarrow j$, and (6) $k \rightarrow j \rightarrow i$. Given a sequencing, $a \rightarrow b \rightarrow c$, a lower bound on the makespan is $h_a + \underline{p}_a + \underline{p}_b + \underline{p}_c + t_c$. In the following let $LB(1), \ldots, LB(6)$ be such lower bounds corresponding to the sequences (1)–(6). New precedence relations may be deduced as follows:

$$\begin{aligned}
&LB(1) \geq UB \wedge LB(2) \geq UB \wedge LB(5) \geq UB \Rightarrow j \rightarrow i \\
&LB(3) \geq UB \wedge LB(4) \geq UB \wedge LB(6) \geq UB \Rightarrow i \rightarrow j \\
&LB(1) \geq UB \wedge LB(2) \geq UB \wedge LB(3) \geq UB \Rightarrow k \rightarrow i \\
&LB(4) \geq UB \wedge LB(5) \geq UB \wedge LB(6) \geq UB \Rightarrow i \rightarrow k \\
&LB(1) \geq UB \wedge LB(3) \geq UB \wedge LB(4) \geq UB \Rightarrow k \rightarrow j \\
&LB(2) \geq UB \wedge LB(5) \geq UB \wedge LB(6) \geq UB \Rightarrow j \rightarrow k
\end{aligned}$$

When new precedence constraints are added, the current solution may become infeasible and needs to be repaired for the search to go on. Sometimes, this results in the new solution having a worse makespan than the solution on the basis of which precedence relations were added, which is inconvenient since the search will continue from this worse solution. It is thus worthwhile to spend some time repairing a solution such that it is at least as good as the original one. To this end, Fleszar and Hindi (2004) construct a special repair algorithm. We take a different approach and use the repair neighborhoods: We construct a partial activity list, and a corresponding set of activities which must be reinserted. The set is constructed as follows: Scan through the activity list of the current solution, if an activity is encountered for which a least one predecessor has not yet been encountered, remove that activity from the list. Each repair neighborhood is in turn given a chance to repair the solution until either a solution which is at least as good as the original is found or there are no neighborhoods left, in which case the repaired solution with the best makespan is used.

Mode diminution As is the case with precedence augmentation, when one finds a new upper bound, UB, one is only interested in subsequent improvements. Thus modes, which lead to a lower bound larger than $UB - 1$ may be removed. When a new UB is found a mode $m \in M_i$ of an activity $i \in \mathcal{A}$, is removed if one of the following is true:

1. $h_i + p_{im} + t_i \geq UB$.
2. $b_{ii} \geq -p_{im} + 1$.
3. The current entries of the FSD-matrix implies i must run in parallel with some activity j, and m is incompatible with all modes of j.

The removal of a mode, may render other modes non-executable, and may render some non-renewable resources redundant. Thus the two first steps of Algorithm 3 are rerun, if a mode is removed.

As for precedence augmentation, the removal of modes may render the current solution infeasible. If this is the case new modes must be selected for the affected activities. We employ a two-step approach: Let $i \in \mathcal{A}$ be some affected activity. First select from among the remaining modes i one that satisfies the non-renewable resources. If no such mode exists, then construct a new schedule in the same way as the initial schedule. If this fails, stop.

Early termination Throughout the course of the algorithm a lower bound is maintained through the entries of the FSD-matrix. If at any point the current upper bound equals the lower bound, the search may be halted, as an optimal solutions has been found. One way to calculate lower bounds is through so-called destructive arguments (see Klein and Scholl (1999)): Assume a minimization problem, with integer-valued objective function $f(x)$, has to be solved, and that a lower bound LB is known. A so-called restricted problem is formed, where the constraint $f(x) \leq LB$ is added. This constraint is then, via the application of so-called reduction algorithms, used to discover new constraints or modify the problem data so as to strengthen the formulation. If the problem is discovered to be infeasible, then the lower bound may be improved to $LB + 1$.

The process of imposing an upper bound of $T = UB - 1$, when a new solutions has been found, updating the entries of the FSD-matrix, and subsequently applying precedence augmentation and mode diminution, can be seen as the application of such reduction algorithms. If through this process the problem is found to be infeasible, the current UB is optimal. The following infeasibility checks are performed:

1. Some $i \in \mathcal{A}$ exists such that $b_{ii} \geq -\underline{p}_i + 1$.
2. Some pair (i, j) exists such that i and j must run in parallel, and this is not feasible for any combination of modes.
3. Some triple (i, j, k) exists such i, j and k must be run in parallel, and this is not feasible for any combination of modes.

One could also perform additional infeasibility checks, such as considering even larger sets of activities, but there is a trade-off between the gain of early termination and the time spent performing these tests.

Justification Valls et al. (2004) shows that a simple technique denoted *justification* can be easily added to algorithms for the RCPSP, improving the quality of the solution with little extra computational effort. One speaks of *left-justification*, *right-justification* and *double-justification*. Left-justification is essentially pulling all activities of a schedule as far towards time zero (left) as possible, while right-justification is essentially pulling all activities as far towards the time corresponding to the makespan (right) as possible. Double-justification consists of a right-justification followed by a left-justification. Often a double-justified schedule is better than the original one. Since this is a simple technique, which has been shown to produce good results, all schedules

produced during the course of the ALNS algorithm are double-justified. This means that in each iteration at least three schedules are generated (more may be generated if the current solution must be repaired after precedence augmentation).

Opportunistic mode-flipping As for justification, *opportunistic mode-flipping* is a simple technique, which is applied after the generation of a new schedule, and which may result in an improvement. Let $i \in \mathcal{A}$ be some activity, and let $m \in \mathcal{M}_i$ be the mode in which i is scheduled. Let $m' \neq m$ be some other mode of i. If changing the mode of i from m to m' does not result in an increase of the makespan, we denote the change a *makespan-preserving mode-flip.*

Proposition 1. *Let $m(i)$ be the current mode of an activity $i \in \mathcal{A}$ for the schedule S. Let σ_i be the starting time of i, let $\tau_i = \min\{\sigma_j | \, j \in \mathcal{S}_i\}$, be the earliest starting time of a successor of i, and let $R(t_0, t_1)_{ik} = \min\{R_k - \sum_{j \in \mathcal{A}(t) \setminus \{i\}} r_{jkm(j)} | \, t \in [t_0, t_1]\}$, be the minimum amount of capacity left in the interval $[t_0, t_1]$ on the resource $k \in \mathcal{R}$. Let $m' \neq m(i)$ be some other mode of i. If $R(\sigma_i, \sigma_i + p_{im'})_{ik} \geq r_{ikm'} \, \forall k \in \mathcal{R}$ and $\sigma_i + p_{im'} \leq \tau_i$, then changing m to m' is a makespan-preserving mode-flip.*

Proof. $R(\sigma_i, \sigma_i + p_{im'})_{ik} \geq r_{ikm'} \, \forall k \in \mathcal{R}$ ensures that no activity scheduled in parallel with i after the flip, has to be moved earlier or later. The only other effected activities could be successors of i, but $\sigma_i + p_{im'} \leq \tau_i$ ensures these do not have to be moved either. □

Opportunistic mode-flipping is applied every time a new schedule is generated: In the order defined by the activity list each activity is examined to see if a makespan-preserving mode-flip may be performed. If so, it is done. If more than one makespan-preserving mode-flip exists for the same activity, the one resulting in the shortest processing time is performed.

The procedure can be seen as an extension of the single-pass improvement method used by Hartmann (2001) for his genetic algorithm. The single-pass improvement method is itself based on the definition of the so-called multi-mode left shift bounding rule originally used in an exact algorithm, see Sprecher et al. (1997). The difference between the procedure of Hartmann (2001) and the procedure proposed here is that for the former only mode-flips, which results in shorter processing times are performed, while this is not the case for the latter.

Overview An overview of the algorithm can be seen in Algorithm 4, where x indicates an activity list, while S_x is the associated schedule after it has been generated and $|S_x|$ is the makespan. Lines 1–2 corresponds to the initial mode and resource reductions, and the initial resource strengthening. Lines 5–11 corresponds to the propagation of lower bounds, and subsequent application of precedence augmentation and mode diminuation. Lines 13–15 is where the mode-change algorithm is called. On Line 16 the destroy and repair neighborhoods are chosen, and a new activity list is created. Finally on Line 18 a new schedule is generated and double justification and opportunistic mode-flipping is applied.

6 Neighborhoods

An important part of an ALNS algorithm are the destroy and repair neighborhoods. Even though there is an adaptive layer, one should remain careful about adding too many neighborhoods, especially when only a limited number of iterations is allowed. The reason is that a number of iterations will be needed before any neighborhoods performing poorly on the current instance will have been filtered out, and these neighborhoods will end up taking time from the good ones. It is also important to have a good mix of neighborhoods, which are good at search intensification and diversification. In the following we describe the destroy and repair neighborhoods employed.

Destroy neighborhoods Given the parameter Q and an activity list, L, a destroy neighborhood must remove Q activities from L. Some of the destroy neighborhoods share the same structure: A predicate function marks activities from L. The marked activities are then removed at random

Algorithm 4 Pseudocode for the ALNS algorithm

Require: Initial values of Q and c, the initial scores $\{\pi_j\}$ and the #schedules max.

1: Remove non-executable and inefficient modes and redundant non-renewable resources.
2: Strengthen resource constraints.
3: $s \leftarrow 0, \quad LB \leftarrow$ lower bound.
4: Create an initial solution S_{x^*}.
5: Do precedence augmentation and mode diminution using $|S_{x^*}|$ as upper bound. If this results in a better lower bound store it in LB.
6: **if** x^* is no longer precedence feasible **then**
7: Repair x^* using the repair neighborhoods and store the best found schedule in S_x. If this results in a better makespan then store it as S_{x^*}.
8: **end if**
9: **if** x^* is no longer mode feasible **then**
10: Find new mode-selection for x^*. If this results in a better makespan then store it as S_{x^*}.
11: **end if**
12: **while** $s < max$ and $|S_{x^*}| > LB$ **do**
13: **if** iterations since last run of mode-change-algorithm is large enough **then**
14: Run mode-change-algorithm
15: **end if**
16: Choose a destroy and repair neighborhood (N^-, N^+) based on $\{\pi_j\}$ and create a new candidate activity list x' from the current schedule S_x.
17: **if not** `Tabu`(x') **then**
18: Create a new candidate schedule $S_{x''}$ from x' using serial SGS, double justification, and opportunistic mode-flipping.
19: **if** $|S_{x''}| < |S_{x^*}|$ **then**
20: $S_{x^*} \leftarrow S_{x''}$
21: Repeat the procedure from line 5 – 11 with the new best solution S_{x^*}.
22: **else if** $|S_{x''}| \leq |S_x|$ **then**
23: $S_x \leftarrow S_{x''}$
24: **end if**
25: **end if**
26: $Q \leftarrow c \cdot Q$
27: Update the number of generated schedules s.
28: Update the scores π_j for N^- and N^+
29: **end while**
30: **return** S_{x^*}

from L along with zero or more activities relating to the ones removed, until Q activities are removed, or no more marked activities are left. More formally: For $j \in \mathcal{A}$ the *predecessor-cluster* of j is defined as $C^p(j) = \{i \in \mathcal{A} | i \in \mathcal{P}_j \vee \sigma_i + p_{im(i)} = \sigma_j\}$, and the *cluster* of j as $C^c(j) = \{i \in \mathcal{A} | i \in C^p(j) \vee i \in \mathcal{S}_j \vee \sigma_j + p_{jm(i)} = \sigma_i\}$, where σ_j denotes the starting time of activity j. Let $p : \mathcal{A} \rightarrow \{0, 1\}$ be some predicate, then a *core removal candidate set*, C, is constructed as $C = \{j \in \mathcal{A} | p(j) = 1\}$. At random an activity j from C is removed from the list along with elements from either $C^p(j)$ or $C^c(j)$ (depending on the neighborhood) until either the set C is empty or Q elements have been removed from L.

The idea behind clusters is that one wants as much flexibility as possible for the repair neighborhood, e.g., if all the predecessors and successors of an activity are left in place there is potentially little room for inserting the activity in new positions.

There are in total 10 destroy neighborhoods, which are described below.

- **random (two flavors)** This neighborhood ensures diversification by randomly removing Q activities from the current solution. It comes in two flavors, one where predecessor-clustering is used and one where clustering is used.

- **most-mobile** Let $j \in \mathcal{A}$. As in Fleszar and Hindi (2004), we define the *left limit* $LL(j)$ and the *right limit* $RL(j)$ as $LL(j) = \max\{\gamma_i | i \in P_j\} + 1$ and $RL(j) = \min\{\gamma_i | i \in S_j\} - 1$, where γ_i is the position of i within the activity list. Now, the *mobility*, $mob(j)$, of j is defined as $mob(j) = RL(j) - LL(j)$.

  This neighborhood selects the Q activities with the highest mobility from the current activity list and also ensures diversification, but in a different way than the random neighborhood described above. It ensures that the neighborhood explored is large by selecting activities which have many reinsertion possibilities.

- **non-peak (two flavors)** In the hybrid genetic algorithm proposed by Valls et al. (2008) the peak crossover operator employed passes on to its children the parts of the schedules with high utilization, so-called *peaks*. Similarly we define a *non-peak* predicate. A peak is defined in the same way as by Valls et al. (2008): Let S be the current schedules at let $S(t) = \{j \in \mathcal{A} | \sigma_j \leq t \wedge t \leq \sigma_j + p_{jm(j)}\}$, where σ_j is the starting time of activity j and t is some point in time. We define the *Resource Utilization Ratio* as follows

$$RUR(t) = \frac{1}{|\mathcal{R}|} \cdot \sum_{j \in S(t)} \sum_{k \in \mathcal{R}} \frac{r_{jk}}{R_k}$$

  Given some $\delta \in [0; 1]$ we say that a point in time, t, is of *high utilization* if $RUR(t) \geq \delta$. Similarly we say that a time interval I is of *high utilization* if $\forall t \in I : RUR(t) \geq \delta$. Let $\mathcal{I}$ be the set of disjunctive maximal intervals of high utilization for the schedule S, then a *peak activity* $j \in \mathcal{A}$ is an activity which satisfies $\exists I \in \mathcal{I} : [\sigma_j; \sigma_j + p_{jm(j)}] \cap I \neq \emptyset$, i.e., all activities, which are active during some interval of high utilization. A *non-peak activity* is an activity which is not a peak activity. The non-peak predicate selects all activities, which are non-peak activities.

  This neighborhood uses the non-peak predicate and tries to preserve the structure of the solution where the utilization is good, and destroy the parts where it is not. The neighborhood comes in two flavors one where predecessor-clustering is used and one where clustering is used. In both cases the activities are chosen at random from the removal candidate set.

- **critical-path (two flavors)** Given the current schedule S we construct a weighted directed graph $G = (\mathcal{A}, E)$, where $E = \{(i, j) \in \mathcal{A} \times \mathcal{A} | \sigma_i + p_{im(i)} = \sigma_j\}$ and the weight of a vertex j is $p_{jm(j)}$. Since the schedule S does not contains "holes" where no activities are in progress, there must exist at least one path with weight equal to the makespan of S. In order to improve the makespan, at least one of the activities on this path must be moved elsewhere. The *critical-path* predicate selects all activities, which are part of a critical path and at random selects Q for removal. The neighborhood comes in two flavors, one where predecessor-clustering is used and one where clustering is used.

- **segment** This neighborhood ensures a certain intensification by selecting a subsequence of length Q from the activity list. This subsequence corresponds to a sub-schedule, which can hopefully be improved.

- **time-windows (two flavors)** Let the head and tail of an activity j be given by h_j and t_j respectively, and let UB be the current upper bound. For any schedule to improve upon this upper bound, j must be scheduled in the interval $[h_j; UB - t_j - (p_{jm(j)} - \underline{p}_j)]$. The time-window predicate selects all activities which are scheduled outside this interval in the current schedule.

  The neighborhood comes in two flavors, one where predecessor-clustering is used and one where clustering is used. The activities from the candidate set are chosen at random. If the candidate set has less than Q elements, additional activities are randomly chosen.

Repair neighborhoods Given a partial activity list and a set of activities to be reinserted a repair neighborhood must construct a new precedence ordered activity list. Again each repair neighborhood shares some structure: Given an ordering of the set of activities to be reinserted, the activities are considered one at a time, and inserted into the activity list such that the activity list is still precedence ordered. Each activity j is inserted randomly within the interval defined by $LL(j)$ and $RL(j)$. A similar move operator was used by Fleszar and Hindi (2004).

There are in total 11 repair neighborhoods, where each neighborhood corresponds to some ordering of the candidates to be reinserted. We employ ordering corresponding to the following well-known priority-rules for the RCPSP (see for instance Kolisch and Hartmann (1999) or Hartmann (1999)): *Shortest processing time* (SPT), *most total successors* (MTS), *earliest start time* (EST), *minimum latest finish time* (LFT), *minimum slack* (MSLK), *greatest rank positional weight* (GRPW), *Weighted Resource Utilization and Precedence* (WRUP) and *minimum latest start time* (LST). These neighborhoods ensure that the solution will be a mix of these priority rules and is in a sense similar to multi-pass methods, where the priority rule is changed between passes.

We define two additional priority rules based on the *volume* of an activity: For some $j \in \mathcal{A}$, we define the volume of j as $v(j) = p_{jm(j)} \cdot \prod_{r \in \{\tilde{r}_{jkm(j)} \mid \tilde{r}_{jkm(j)} > 0, k \in \mathcal{R}\}} r$. The first priority rule called *smallest volume first* (SVF) orders the activities non-decreasingly w.r.t volume, while the second priority rule called *largest volume first* (LVF) orders the activities non-increasingly w.r.t. volume.

The two last orderings, are the random ordering (RAN), and the reverse (REV) ordering, which reverses the order of the activities in the current activity list.

7 Computational experiments

In this section we present the computational experiments performed. This includes the quality and effect of the lower bounds and the effects of the different components of the proposed algorithm. We conclude with a comparison to other algorithms for the MRCPSP. The algorithm has been coded in C++, compiled with gcc 4.4.3 and the experiments have been run on a PC with 2 Intel(R) Xeon(R) CPU X5550 @ 2.67GHz (16 cores in total, but only a single core is used), with 24 GB of RAM, and running Ubuntu 10.4. The code is available for download at `http://diku.dk/~laurent`.

The experiments have been performed on the well-known MRCPSP benchmark classes, J10, J12, J14, J16, J18, J20 and J30 available at `http://129.187.106.231/psplib/`. The benchmark classes consists of instances containing respectively 10, 12, 14, 16, 18, 20, and 30 non-dummy activities. Each activity may be performed in up to 3 different modes, there are two non-renewable, and two renewable resources. The number of feasible instances in each benchmark class J10–J20 is respectively 536, 547, 551, 550, 552, and 554. For these classes all optimal solutions are known. This is not the case for J30. Here there are in total 640 instances of which 552 are known to be feasible.

In order to establish whether the remaining 88 instances of J30 are in fact infeasible, we construct an Interger Programming (IP) containing only the non-renewable resource costraints, and use ILOG CPLEX 12.1 to solve the problems. This confirms that these 88 instances are infeasible. As a result, when experimenting on this benchmark class J30, the algorithm is only run on the feasible instances.

Each experiment has been repeated 10 times and the average is reported. Traditionally the maximum number of generated schedules is used as stopping criteria in order to make it easy to compare algorithms. One generated schedule should correspond to assigning all activities a starting time, as is done by a pass using the serial or parallel SGS. For the ALNS algorithm we count 1 schedule for every translation of the activity list into a schedule, 2 additional schedules for the application of double justification, and finally 1 additional schedule for the application of opportunistic mode-flipping. Unless otherwise specified 5000 generated schedules is used as stopping criteria.

The following parameters are set as found by Muller (2009): score reaction = 0.2, score interval = 5, initial $Q = 40\%$, minimal score = 10^{-4}, and cooling coefficient c set such that Q reaches a percentage corresponding to one activity after the specified number of generated schedules.

7.1 Lower bounds

We here examine the quality of the lower bounds, and the time spend to calculate them on the J10, J20, and J30 benchmark classes. Each bound is initially tested individually. Based on these tests, two groups of bounds are made, and additional experiments performed.

The results of the initial experiment can be seen in Table 1. As can be seen, for all classes the bound LBX2 performs the best. However, LBX2 has a large computational cost. LBX1 is relatively close to LBX2 while taking considerably less time. We will in the following use LBX1, and exclude LBX2.

Using LB2X to replace LB2 as a stand-alone bound improves the bound, and does not result in a significant increase in computational time (around a factor 2), given the relatively low computational times (in the order of seconds in total for around 550 instances). Using the simpler bound LB2X' to replace LB2 as a stand-alone bound, again improves the bound, while the computational time is unchanged compared to LB2 However, the bound improvement is less than when using LB2X, which is as expected, and in the following we use LB2X as a stand-alone bound.

Using resource strengthening as expected improves the bound for LB2, LB2X, and LB2X'. Most significantly on the smaller J10 instance. In the following we thus enable resource strengthening.

Using the bound LB2X instead of LB2 for the inner subproblems is effective, yielding improved bounds in all cases, although in many cases the improvement is slight. The increase in computational time from using LBX2 instead of LB2 could be deemed acceptable for bounds other than LBX1 and LBX2, where it is prohibitively large. The reason is that more inner subproblems are evaluated for LBX1, and LBX2 than for the remaining. Using the simpler bound LB2X' gives results relatively close to using LB2X but using less computational time.

To further test the bounds we create two groups: LBS contains LB1, LB2X, LB6, LB8, and LB11, and LBSX1 contains all the bounds of LBS and LBX1. The reason for this experiment, is that we want to see whether it is worth including the time-consuming bound LBX1, and whether it is worth using LB2X, or LB2X' when evaluating inner subproblems. The results from comparing these two groups can be seen in Table 2. As can be seen including LBX1 does improve the bounds, though as expected at an increased computational time. Using LB2X, and LB2X' (rather than LB2) for the inner subproblems also improves the bounds, but only slightly. Again the running time increases. However, the increase is much smaller for LB2X' than for LB2X. We deem that the improvement in quality from including LBX1 is worth the increase in running time, while we choose not to use LB2X, nor LB2X' when calculating bounds of inner subproblems, as the improvement in quality is only slight compared to the additional running time incurred. In the sequel, unless otherwise stated we thus employ LBSX1, with LB2 for inner subproblems.

7.2 Components

We here examine the effects of the different components of precedence augmentation, mode-diminution, opportunistic mode-flipping, tabu-list, scoring, and bounds in order to judge their effectiveness. For each of the experiments, all components are enabled, except for the one being examined.

Precedence augmentation We first examine the effect of precedence augmentation. For each of the J10, J20, and J30 benchmark classes we make two experiments, with and without precedence augmentation enabled. The results can be seen in Table 3. Precedence augmentation does not appear to have a significant impact on the quality of the solutions. We believe that the reason for this is that even though the search space is narrowed, the algorithm may continue from a worse solution than the current best, because the addition of precedence relations has rendered the current best solution infeasible. This means that fewer iterations will be used for searching neighborhoods of best solution.

Mode-diminution We next examine the effect of mode-diminution. We proceed as earlier and make two different experiments, with and without mode-diminution enabled. The results can be

Table 1: Quality of different lower bounds, and the time spend. The **Avg.** column is the average relative distance in percent to the critical path lower bound. Thus higher is better. The **T. Time** column is the total time used in seconds to calculate the bound accumulated across all the instances of a class. LB2X', and LB2X indicates that the respective bound is used instead of LB2 when calculating bounds on inner subproblems (empty for bounds which do not use inner subproblems, and for LB2 it means that this bound is replaced by the respective bound). For each class **boldfont** indicates the best bound, while for each bound an underline indicates the best result (where applicable). An asterix (*) means that resource strengthening has not been used.

		Normal		LB2X'		LB2X	
	Bound	**Avg.(%)**	**T. time(s)**	**Avg.(%)**	**T. time(s)**	**Avg.(%)**	**T. time(s)**
J10	LB1	0.00	0.1	–	–	–	–
	LB2	4.99	0.1	5.14	0.1	5.30	0.2
	LB2*	3.79	0.1	3.93	0.1	4.05	0.2
	LB6	5.78	0.2	5.84	0.1	5.98	0.4
	LB8	3.58	0.1	–	–	–	–
	LB10	5.12	0.1	5.25	0.1	5.35	0.2
	LB11	5.45	0.1	5.58	0.1	5.70	0.2
	LBX1	5.87	0.3	5.91	0.5	5.92	6.7
	LBX2	5.95	1.3	5.98	2.6	**5.99**	34.7
J20	LB1	0.00	0.1	–	–	–	–
	LB2	1.81	0.4	1.91	0.4	1.97	0.9
	LB2*	1.79	0.4	1.90	0.4	1.96	0.9
	LB6	1.36	1.0	1.42	1.1	1.47	4.7
	LB8	0.45	0.9	–	–	–	–
	LB10	1.62	0.4	1.70	0.4	1.73	0.8
	LB11	1.97	0.6	2.06	0.7	2.11	1.1
	LBX1	2.14	6.1	2.18	10.9	2.20	139.8
	LBX2	2.22	66.3	**2.24**	128.1	**2.24**	1571.9
J30	LB1	0.00	0.2	–	–	–	–
	LB2	2.11	0.9	2.21	0.9	2.25	2.1
	LB2*	2.10	0.7	2.20	1.0	2.24	2.1
	LB6	1.29	3.1	1.33	3.5	1.36	17.0
	LB8	0.04	2.4	–	–	–	–
	LB10	1.12	0.9	1.18	1.0	1.21	2.2
	LB11	2.11	2.1	2.21	2.1	2.25	3.4
	LBX1	2.28	31.1	2.35	54.2	2.38	649.2
	LBX2	2.31	539.1	2.36	918.8	**2.39**	10875.9

seen in Table 4. The reason why there are still modes removed when mode-diminution is disabled is that the initial mode removal is still run. As can be seen mode-diminution gives an improvement in solution quality. We judge the increase in computational time is worth the improved solutions, and include mode-diminution for the final results.

Opportunistic mode-flipping We next examine the effect of opportunistic mode-flipping. We proceed as earlier and make two different experiments, with and without opportunistic mode-flipping enabled. As this procedure can be seen as an extension of the single-pass improvement method used by Hartmann (2001) it would be of interest to compare to this procedure also. We thus perform an experiment, where mode-flipping only occurs, if the processing time is reduced. This should be equivalent to the procedure used in Hartmann (2001).

The results can be seen in Table 5. Opportunistic mode-flipping has a significant effect on the quality of the solutions. This effect gets more pronounced as the size of the instances grow. The

Table 2: Quality of groups of lower bounds, and the time spend. *LBS* contains LB1, LB2X, LB6, LB8 and LB11. LBSX1 in addition contains LBX1. The columns are equivalent to those of Table 1.

		Normal		LB2X'		LB2X	
	Bound	**Avg.(%)**	**T. time(s)**	**Avg.(%)**	**T. time(s)**	**Avg.(%)**	**T. time(s)**
J10	LBS	6.47	0.4	6.48	0.5	6.49	0.8
	LBSX1	6.90	0.8	6.92	1.0	**6.94**	7.8
J20	LBS	2.22	3.1	2.24	3.2	2.24	7.6
	LBSX1	2.39	9.5	2.43	14.8	**2.45**	152.9
J30	LBS	2.26	9.4	2.27	9.8	2.27	25.2
	LBSX1	2.34	40.6	2.37	63.9	**2.40**	670.8

Table 3: Shows effect of precedence augmentation. **Avg.** is the average relative distance in percent to the critical path lower bound, **Dev.** is the standard deviation of this value across the 10 runs which constitute the experiment, **Added** is the average number of precedence relations added, and **Time** is the average time used per instance. **boldface** indicates the best average relative distance for each benchmark class.

Precedence augmentation							
	Without			**With**			
	Avg.(%)	**Dev.**	**Time(s)**	**Avg.(%)**	**Dev.**	**Added**	**Time(s)**
J10	32.36	0.036	0.03	**32.32**	0.029	1.08	0.03
J20	**19.12**	0.084	0.09	19.13	0.067	5.93	0.11
J30	**16.01**	0.080	0.34	**16.01**	0.101	16.11	0.40

Table 4: Shows effect of mode-diminution. The columns **Avg.**, **Dev.** and **Time**, and **boldface** notation is as earlier. The **Rem.** column is the average number of modes removed.

Mode diminution								
	Without				**With**			
	Avg.(%)	**Dev.**	**Rem.**	**Time(s)**	**Avg.(%)**	**Dev.**	**Rem.**	**Time(s)**
J10	32.86	0.055	1.99	0.04	**32.35**	0.055	7.62	0.03
J20	20.25	0.143	2.62	0.11	**19.12**	0.092	9.95	0.11
J30	16.86	0.101	3.52	0.34	**15.98**	0.086	10.71	0.42

total time used is also decreased when using this technique, which is explained by the fact that the upper bound is better, and thus more often the algorithm may stop early because the lower bound has been reached.

Opportunistic mode-flipping improves upon the results of the Hartmann (2001)–equivalent procedure, which justifies the extension. Opportunistic mode-flipping is not tied to the algorithm itself, and could be plugged into other algorithms for the MRCPSP, perhaps resulting in improved results, at very little extra work. We include this technique for the final results.

Table 5: Shows the effect of opportunistic mode-flipping. In the last group (**With – only shorter**), mode-flipping only occurs if the processing time is reduced. The columns **Avg.**, **Dev.** and **Time**, and **boldface** notation is as earlier.

	Opportunistic mode-flipping								
	Without			**With**			**With – only shorter**		
	Avg.(%)	**Dev.**	**Time(s)**	**Avg.(%)**	**Dev.**	**Time(s)**	**Avg.(%)**	**Dev.**	**Time(s)**
J10	33.03	0.113	0.03	**32.35**	0.040	0.03	32.44	0.659	0.03
J20	24.67	0.169	0.14	**19.13**	0.071	0.11	20.19	0.146	0.12
J30	24.28	0.164	0.59	**16.03**	0.076	0.42	17.88	0.153	0.45

Tabu-list We next examine the effect of the tabu-list. We proceed as earlier and make two different experiments, with and without the tabu-list enabled. The results can be seen in Table 6. The tabu-list has a slightly positive effect. The difference in time consumption is very low. We thus enable the tabu-list for the final results.

Table 6: Shows the effect of the tabu-list. The columns **Avg.**, **Dev.** and **Time**, and **boldface** notation is as earlier.

	Tabu list					
	Without			**With**		
	Avg.(%)	**Dev.**	**Time(s)**	**Avg.(%)**	**Dev.**	**Time(s)**
J10	32.36	0.034	0.02	**32.33**	0.039	0.03
J20	19.18	0.085	0.11	**19.16**	0.092	0.11
J30	16.07	0.137	0.41	**16.04**	0.078	0.42

Adaptive layer We next examine the effect of the adaptive layer. We proceed as earlier and make two different experiments, in the first experiment neighborhoods are picked at random disregarding how they have performed, while in the second experiment the scoring scheme is enabled. As 5000 schedules, will only result in around 1500-2000 iterations of the of the ALNS algorithm, which may not be enough for the addaptive layer to stabilize, we additionally peform a run with 500000 schedules. The results can be seen in Table 7. The adaptive layer, disappointingly, has very slight impact on the solutions.

Bounds We finally examine the effect of using strong bound arguments, as opposed to the simple critical path lower bound. We proceed as earlier and make two different experiments, in the first experiment the algorithm is run using the set of bounds LBSX1, while in the second experiment only the critical path lower bound (LB1) is used. The results can be seen in Table 8. As can be seen a stronger bound argument, does not appear to have a significant impact on the quality of the solutions, nor on the number of precedence relations added, nor the number of modes removed. Even though there is an increase in computational time, we include the stronger bound set for the final results, as there is an improvement for two of the benchmark classes, though slight.

7.3 Final results

In this section we make the final experiments, and compare the results to other heuristics from the literature. To recap, for these final experiments: the bound group LBSX1 is used with LB2 for

Table 7: Shows the effect of the adaptive layer. The columns **Avg.**, **Dev.** and **Time**, and **boldface** notation is as earlier.

		Adaptive layer					
		Without			**With**		
#Sch.		**Avg.(%)**	**Dev.**	**Time(s)**	**Avg.(%)**	**Dev.**	**Time(s)**
5000	J10	32.34	0.035	0.03	**32.32**	0.039	0.03
	J20	19.11	0.089	0.11	**19.07**	0.075	0.11
	J30	16.04	0.093	0.42	**16.00**	0.106	0.42
500000	J10	**32.26**	0.022	2.9	**32.26**	0.025	2.8
	J20	**18.14**	0.051	6.4	18.18	0.063	6.4
	J30	14.62	0.040	13.1	**14.60**	0.065	13.2

Table 8: Effect of using strong lower bound arguments as opposed to the critical path lower bound. The column **LBSX1** are the results with the LBSX1 bounds enabled, while the column **LB1** are the results with the critical path lower bound alone. The rows indicate respectively the average deviation from the critical path lower bound, the average number of precedence relations added, the average number of modes removed, and average time used per instance. **boldface** indicates the best result for each line and benchmark class.

	J10		J20		J30	
	LBSX1	**LB1**	**LBSX1**	**LB1**	**LBSX1**	**LB1**
Avg. dev.(%)	**32.33**	32.36	19.14	**19.12**	**16.00**	16.01
Prec. added	**1.08**	0.97	**5.93**	5.75	**16.79**	15.81
Modes rem.	**7.60**	7.25	**9.90**	9.74	10.56	**10.64**
Time(s)	**0.03**	**0.03**	0.11	**0.07**	0.42	**0.19**

the inner subproblems, resource strengthening and mode-removal is applied initially, precedence augmentation is disabled, and mode-diminution, opportunistic mode-flipping, the tabu-list, and the adaptive layer is enabled.

The results from running the algorithm on the benchmark classes for different values of the number of maximum generated schedules can be seen in Table 9. Table 10 gives a comparison to other heuristics for the MRCPSP found in the literature on the benchmark classes J10-J20 for 5000 schedules. Table 11 gives a comparison to other heuristics for the MRCPSP on the benchmark class J30 for 5000 schedules. As not all solutions to this benchmark class are known to optimality, we only state the average relative deviation to the critical path lower bound. Not all heuristics state their results as a function of the number of generated schedules. To be able to compare with these, we do as Lova et al. (2009). The results can be seen in Table 12. We note that we beat both the algorithm of Lova et al. (2009) and Van Peteghem and Vanhoucke (2009) given 50.000 schedules (0.5 seconds of computational time per instance).

New best solutions During the experiments new best solutions were found for the following instances of the J30 benchmark class (compared to those reported by the PSPLIB): `j3013_4` (new solution 41, previous was 42), `j3046_7.mm` (new solution 43, previous was 44), and `j3047_7.mm` (new solution 28, previous was 29).

Table 9: Results on benchmark instances J10-J20, and J30 for different number of schedules. The **Avg.** column is the average relative distance in percent to the optimal solutions for J10–J20, and for J30 it is the average relative deviation from the critical path lower bound. **Opt.** is the percentage of instances solved to optimality for J10–J20, and for J30 it is the number of instances with solutions equal to the best known. **Time** is the average time used per instance.

	#Sch.	**Avg.(%)**	**Opt.(%)**	**Time(s)**		**#Sch.**	**Avg.(%)**	**Opt.(%)**	**Time(s)**
	1000	0.26	95.56	0.01		1000	0.56	89.89	0.01
	3000	0.09	98.41	0.02		3000	0.26	94.7	0.02
J10	5000	0.05	99.01	0.03	J12	5000	0.20	95.59	0.03
	6000	0.05	99.09	0.03		6000	0.18	96.07	0.04
	50000	0.02	99.76	0.50		50000	0.11	97.77	0.43
	500000	0.02	99.72	2.90		500000	0.06	98.72	5.84
	1000	1.13	79.24	0.03		1000	1.45	74.49	0.06
	3000	0.70	85.99	0.05		3000	0.93	82.07	0.07
J14	5000	0.55	88.75	0.08	J16	5000	0.78	84.76	0.09
	6000	0.50	89.40	0.07		6000	0.71	86.35	0.10
	50000	0.28	94.03	0.39		50000	0.48	91.16	0.46
	500000	0.21	95.86	5.96		500000	0.40	92.40	4.55
	1000	1.89	69.84	0.24		1000	2.32	65.32	0.05
	3000	1.28	76.30	0.31		3000	1.72	70.72	0.07
J18	5000	1.14	78.75	0.31	J20	5000	1.52	72.87	0.10
	6000	1.06	79.46	0.32		6000	1.46	73.57	0.11
	50000	0.77	85.89	0.78		50000	1.05	81.06	0.61
	500000	0.62	89.09	5.53		500000	0.85	85.24	6.34
	1000	17.04	58.95	0.20					
	3000	16.32	61.68	0.27					
J30	5000	15.96	62.97	0.34					
	6000	15.95	63.06	0.37					
	50000	15.11	67.74	1.63					
	500000	14.58	72.81	13.67					

8 Conclusion

We have presented an ALNS-based algorithm for the MRCPSP. As part of this algorithm we have proposed three new multi-mode specific bounds for the MRCPSP: one (LB2X) is based on a Lagrange relaxation and is an extension of the capacity bound LB2, while the two others (LBX1 and LBX2) are based on testing all combination of mode assignments for respectively pairs and triples of activities. A lightweight version of LB2X, LB2X', was also examined. Computational experiments have shown that these bounds are an improvement over existing bounds found in the literature (which do not take multiple modes into account), but take additional computation time.

The algorithm employs a number of different techniques for reducing the search space. Again computational experiments have shown that some of these techniques (mode-diminuation and opportunistic mode-flipping) are effective, while others (precedence augmentation, the adaptive layer and the tabu-list) do not have a significant impact. For the techniques that employ lower bound arguments (precedence augmentation and mode-diminuation), we have investigated the effect of the quality of lower bound on the effectiveness of the techniques. Surprisingly it turns out that the quality of the bound argument only has a very small influence.

Two relatively simple techniques, opportunistic mode-flipping and mode diminution, proved to be effective. An interesting area of research would be to investigate whether these techniques could be beneficially incorporated into other heuristics for the MRCPSP.

Experiments on a set of standard benchmark instances from the literature have shown that the ALNS algorithm is competitive with other state-of-the-art heuristics, but can not beat the best.

Table 10: Comparison to other heuristics on the benchmark class J10-J20, and for 5000 schedules. The **Avg.** column is the average relative distance in percent to the optimal solutions, and **Opt.** is the percentage of instances solved to optimality

J10	**Avg. (%)**	**Opt. (%)**
Van Peteghem and Vanhoucke (2009)	0.02	99.44
ALNS	0.05	99.01
Lova et al. (2009)	0.06	98.51
Chiang et al. (2008)	0.16	–
Ranjbar et al. (2009)	0.18	–
Alcaraz et al. (2003)	0.24	–
Tseng and Chen (2009)	0.33	95.16
Józefowska et al. (2001)	1.16	–

J12	**Avg. (%)**	**Opt. (%)**
Van Peteghem and Vanhoucke (2009)	0.07	98.35
Lova et al. (2009)	0.17	96.53
ALNS	0.20	95.59
Tseng and Chen (2009)	0.52	90.57
Ranjbar et al. (2009)	0.65	–
Alcaraz et al. (2003)	0.73	–
Józefowska et al. (2001)	1.73	–

J14	**Avg. (%)**	**Opt. (%)**
Van Peteghem and Vanhoucke (2009)	0.20	95.10
Lova et al. (2009)	0.32	92.92
ALNS	0.55	88.75
Ranjbar et al. (2009)	0.89	–
Tseng and Chen (2009)	0.92	82.03
Alcaraz et al. (2003)	1.00	–
Józefowska et al. (2001)	2.60	–

J16	**Avg. (%)**	**Opt. (%)**
Van Peteghem and Vanhoucke (2009)	0.39	90.36
Lova et al. (2009)	0.44	90.00
ALNS	0.78	84.76
Ranjbar et al. (2009)	0.95	–
Tseng and Chen (2009)	1.09	77.39
Alcaraz et al. (2003)	1.12	–
Józefowska et al. (2001)	4.07	–

J18	**Avg. (%)**	**Opt. (%)**
Van Peteghem and Vanhoucke (2009)	0.52	86.23
Lova et al. (2009)	0.63	84.96
ALNS	1.14	78.75
Ranjbar et al. (2009)	1.21	–
Tseng and Chen (2009)	1.30	73.38
Alcaraz et al. (2003)	1.43	–
Józefowska et al. (2001)	5.52	–

J20	**Avg. (%)**	**Opt. (%)**
Van Peteghem and Vanhoucke (2009)	0.70	81.59
Lova et al. (2009)	0.87	80.32
Chiang et al. (2008)	1.44	–
ALNS	1.52	72.87
Ranjbar et al. (2009)	1.64	–
Tseng and Chen (2009)	1.71	66.66
Alcaraz et al. (2003)	1.91	–
Józefowska et al. (2001)	6.74	–

Table 11: Comparison to other heuristics on the benchmark class J30, and for 5000 schedules. The **Avg.** column is the average relative distance in percent to the optimal solutions.

J30	**Avg.(%)**
Van Peteghem and Vanhoucke (2009)	11.90
Lova et al. (2009)	14.77
ALNS	15.96
Tseng and Chen (2009)	18.32

We were able to find a new best solutions for 3 the benchmark instances.

Table 12: Comparison to other heuristics on the benchmark class J10. The columns **Avg.**, **Opt.**, and **Time** are as earlier, while **#Schedules** is the maximum number of schedules for the algorithms (where applicable).

J10	**Avg.(%)**	**Opt.(%)**	**#Schedules**	**Time(s)**
ALNS	0.02	99.76	50,000	0.50s
Lova et al. (2009)	0.04	99.07	6,000	0.10s[a]
ALNS	0.05	99.09	6,000	0.03s
Hartmann (2001)	0.10	98.10	–	–
Alcaraz et al. (2003)	0.19	96.50	6,000	0.19s[b]
Bouleimen and Lecocq (2003)[c]	0.21	96.30	–	19.3s[c]
Kolisch and Drexl (1997)	0.50	91.80	6,000	–
Ozdamar (1999)	0.86	88.10	6,000	–

[a] Pentium with 3.0 GHz and 1 Gbytes of RAM.
[b] Pentium III with 1.13 GHz and 256 MB RAM.
[c] Pentium with 100 MHz and 32 Mbytes of RAM.

References

Alcaraz, J., Maroto, C., Ruiz, R. Solving the multi-mode resource-constrained project scheduling problem with genetic algorithms. *Journal of the Operational Research Society*, 54(6):614–626, 2003.

Bartusch, M., Möhring, R., Radermacher, F. Scheduling project networks with resource constraints and time windows. *Annals of Operations Research*, 16(1):199–240, 1988.

Blażewicz, J., Lenstra, J., Kan, A. R. Scheduling subject to resource constraints: Classification and complexity. *Discrete Applied Mathematics*, 5:11–24, 1983.

Boctor, F. F. A new and efficient heuristic for scheduling projects with resource restrictions and multiple execution modes. *European Journal of Operational Research*, 90(2):349 – 361, 1996.

Bouleimen, K., Lecocq, H. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple mode version. *European Journal of Operational Research*, 149(2):268–281, 2003.

Brucker, P., Knust, S. A linear programming and constraint propagation-based lower bound for the rcpsp. *European Journal of Operational Research*, 127(2):355–362, 2000.

Brucker, P., Drexl, A., Möhring, R., Neumann, K., Pesch, E. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1):3–41, 1999.

Chiang, C., Huang, Y., Wang, W. Ant colony optimization with parameter adaptation for multi-mode resource-constrained project scheduling. *Journal of Intelligent and Fuzzy Systems*, 19(4): 345–358, 2008.

Damak, N., Jarboui, B., Siarry, P., Loukil, T. Differential evolution for solving multi-mode resource-constrained project scheduling problems. *Computers & Operations Research*, 36(9): 2653 – 2659, 2009.

Demassey, S., Artigues, C., Michelon, P. Constraint-propagation-based cutting planes: An application to the resource-constrained project scheduling problem. *Informs Journal on Computing*, 17:52–65, 2005.

Drexl, A., Gruenewald, J. Nonpreemptive multi-mode resource-constrained project scheduling. *IIE transactions*, 25(5):74–81, 1993.

Fleszar, K., Hindi, K. S. Solving the resource-constrained project scheduling problem by a variable neighbourhood search. *European Journal of Operational Research*, 155(2):402–413, 2004.

Hartmann, S. *Project scheduling under limited resources: models, methods, and applications*, chapter Classification of single-mode heuristics, pages 61 – 81. Springer Verlag, 1999.

Hartmann, S. Project scheduling with multiple modes: A genetic algorithm. *Annals of Operations Research*, 102(1):111–135, 2001.

Hartmann, S., Drexl, A. Project scheduling with multiple modes: A comparison of exact algorithms. *Networks*, 32(4):283–297, 1998.

Hartmann, S., Briskorn, D. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, In Press, Corrected Proof:–, 2009.

Herroelen, W., Demeulemeester, E., Reyck, B. D. Resource-constrained project scheduling - a survey of recent developments. *Computers & Operations Research*, 29(4):279–302, 1998.

Jarboui, B., Damak, N., Siarry, P., Rebai, A. A combinatorial particle swarm optimization for solving multi-mode resource-constrained project scheduling problems. *Applied Mathematics and Computation*, 195(1):299 – 308, 2008.

Józefowska, J., Mika, M., Różycki, R., Waligóra, G., Weglarz, J. Simulated annealing for multi-mode resource-constrained project scheduling. *Annals of Operations Research*, 102(1):137–155, 2001.

Karp, R. Reducibility among combinatorial problems. In Miller, R., Thatcher, J., editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

Klein, R., Scholl, A. Computing lower bounds by destructive improvement: An application to resource-constrained project scheduling. *European Journal of Operational Research*, 112:322–346, 1999.

Kolisch, R., Drexl, A. Local search for nonpreemptive multi-mode resource-constrained project scheduling. *IIE transactions*, 29(11):987–999, 1997.

Kolisch, R., Hartmann, S. Heuristic algorithms for solving the resource-constrained project scheduling problem: Classication and computational analysis. In Weglarz, J., editor, *Project scheduling: Recent models, algorithms and applications*, pages 147–178. Kluwer Academic Publishers, Kluwer, Amsterdam, the Netherlands, 1999.

Kolisch, R., Hartmann, S. Experimental investigation of heuristics for resource-constrained project scheduling. *European Journal of Operational Research*, 127:394–407, 2000.

Kolisch, R., Hartmann, S. Experimental investigation of heuristics for resource-constrained project scheduling: An update. *European Journal of Operational Research*, 174(1):23–37, 2006.

Kolisch, R., Padman, R. An integrated survey of deterministic project scheduling. *Omega*, 29(3): 249–272, 2001.

Lova, A., Tormos, P., Cervantes, M., Barber, F. An efficient hybrid genetic algorithm for scheduling projects with resource constraints and multiple execution modes. *International Journal of Production Economics*, 117(2):302 – 316, 2009.

Mingozzi, A., Maniezzo, V., Ricciardelli, S., Bianco, L. An exact algorithm for the resource-constrained project scheduling problem based on a new mathematical formulation. *Management Science*, 44(5):714–729, 1998.

Mori, M., Tseng, C. C. A genetic algorithm for multi-mode resource constrained project scheduling problem. *European Journal of Operational Research*, 100(1):134 – 141, 1997.

Muller, L. F. An adaptive large neighborhood search algorithm for the resource-constrained project scheduling problem. In *Proceedings of the VIII Metaheuristics International Conference (MIC) 2009*, Hamburg, Germany, 13-16 July 2009.

Okada, I., Zhang, X., Yang, H., Fujimura, S. A random key-based genetic algorithm approach for resource-constrained project scheduling problem with multiple modes. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, volume 1, 2010.

Ozdamar, L. A genetic algorithm approach to a general category project scheduling problem. *IEEE Transactions on Systems, Man and Cybernetics – Part C*, 29(1):44–59, 1999.

Patterson, J. A comparison of exact approaches for solving the multiple constrained resource, project scheduling problem. *Management science*, 30(7):854–867, 1984.

Pisinger, D., Røpke, S. A general heuristic for vehicle routing problems. *Computers & Operations Research*, 34(8):2403–2435, 2007.

Ranjbar, M., Reyck, B. D., Kianfar, F. A hybrid scatter search for the discrete time/resource trade-off problem in project scheduling. *European Journal of Operational Research*, 193(1): 35–48, 2009.

Røpke, S., Pisinger, D. A unified heuristic for a large class of vehicle routing problems with backhauls. *European Journal of Operational Research*, 171(3):750–775, 2006a.

Røpke, S., Pisinger, D. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006b.

Sprecher, A., Hartmann, S., Drexl, A. An exact algorithm for project scheduling with multiple modes. *OR Spectrum*, 19(3):195–203, 1997.

Tchao, C., Martins, S. Hybrid heuristics for multi-mode resource-constrained project scheduling. In Maniezzo, V., Battiti, R., Watson, J.-P., editors, *Learning and Intelligent Optimization*, volume 5313 of *Lecture Notes in Computer Science*, pages 234–242. Springer Berlin / Heidelberg, 2008.

Tseng, C. Two heuristic algorithms for a multi-mode resource-constrained multi-project scheduling problem. *Journal of Science and Engineering Technology*, 4(2):63–74, 2008.

Tseng, L., Chen, S. Two-phase genetic local search algorithm for the multimode resource-constrained project scheduling problem. *Evolutionary Computation, IEEE Transactions on*, 13(4):848–857, 2009.

Valls, V., Ballestín, F., Quintanilla, S. A population-based approach to the resource-constrained project scheduling problem. *Annals of Operations Research*, 131:304–324, 2004.

Valls, V., Ballestín, F., Quintanilla, S. A hybrid genetic algorithm for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 185(2):495–508, 2008.

Van Peteghem, V., Vanhoucke, M. An artificial immune system for the multi-mode resource-constrained project scheduling problem. In Cotta, C., Cowling, P., editors, *Evolutionary Computation in Combinatorial Optimization*, volume 5482 of *Lecture Notes in Computer Science*, pages 85–96. Springer Berlin / Heidelberg, 2009.

Zhang, H., Tam, C., Li, H. Multimode project scheduling based on particle swarm optimization. *Computer-Aided Civil and Infrastructure Engineering*, 21(2):93–103, 2006.

Zhu, G., Bard, J., Yu, G. A branch-and-cut procedure for the multimode resource-constrained project-scheduling problem. *INFORMS Journal on Computing*, 18(3):377, 2006.

We present an Adaptive Large Neighborhood Search algorithm for the Multi-mode Resource-Constrained Project Scheduling Problem (MRCPSP). We incorporate techniques for deriving additional precedence relations and propose a new method, so-called mode-diminution, for removing modes during execution. These techniques make use of bound arguments, and we propose and experiment with three new bounds for the MRCPSP, in addition to bounds found in the literature. We propose a simple technique, so-called opportunistic mode-flipping, which can be applied whenever a schedule is generated,and which significantly improves the results of the algorithm. Computational experiments are performed on a set of standard benchmark instances from the PSPLIB, and a comparison is made with other algorithms found in the literature. The experiments show that the algorithm is competitive, but can not beat the best algorithms. Even so, some of the elements of the algorithm perform well, that is the bound arguments, the mode-removal procedure, and in particular opportunistic mode-flipping, and these elements may perhaps be used to improve the results of other algorithms for this problem.

DTU Management Engineering
Department of Management Engineering
Technical University of Denmark

Produktionstorvet
Building 424
DK-2800 Kongens Lyngby
Denmark
Tel. +45 45 25 48 00
Fax +45 45 93 34 35

www.man.dtu.dk